C++程序设计期末考试-补充 PPT　　2022.11.25

《深入理解 static, const, extern》

static、const 对象的作用域是当前模块（当前.cpp 文件）。不同的模块可以定义同名的 static、const 变量。

1.cpp

```
const int x = 1;
static int y = 2;
static const int z = 3;
const int a = 5;
int k = 0;
```

2.cpp

```
const int x = 1;          //对
static int y = 2;         //对
static const int z = 3;   //对
int k = 0;                //错误
extern const int a;       //对
int main( ) {
   return x + y + z + a;
   //错,找不到a
}
```

变量定义

变量定义（需要分配内存），变量定义的 2 种形式：

- 没有使用 extern 限定，可以不初始化。例如, const int k = 0; int　x, y;　A　a; //A 定义了无参构造函数
- 使用 extern 限定，这时必须初始化（不初始化表明仅仅是声明，这个变量实际上可能不存在）。

extern const int k = 0;　　extern　int　x = 1;

extern　A　a;　　//不是定义，仅是声明

extern　A　a(…);　　//A 定义了有参构造函数

extern　A　a = A();　　//A 定义了无参构造函数

变量定义有 3 种形式:

- 只定义不声明、先声明后定义、先定义后声明

extern 和 static 不能连用。

例：int　x = 1;

extern　int　x, y;　　static　int　y = 2;　　const　int　z = 1;　　extern const　int　z;　　int　m, n;

extern int m;

- 在一个模块内部，一般只定义变量（不声明）。既定义又声明没有意义。
- 只有对于 const [static] 变量，先声明后定义才有意义（先定义后声明　⇔　只定义不声明），表明将该变量的作用域修改为整个程序。

变量声明 extern

- 不给变量分配内存。用于声明可能有这个变量（不管这个变量实际上是否存在）。
- 编译器遇到语句 extern 变量 时，如果程序并没有访问这个变量，则不做任何处理。否则：先在当前模块查找该变量的定义；若没找到则在其他模块中查找该变量的定义，若找到且其作用域是整个程序则成功（否则报错）。

1.cpp

```
int k = 0;
const int x = 1;
extern const int x;
extern const int y;
const int y = 2;
extern const int z;
static const int z = 3;
extern int m;
static int m = 3;
```

2.cpp

```
extern const int x;
extern const int y;
extern const int z;
extern int m;
extern int k = 1; //错, k已经定义
int main( ) {
   return x + y + z + m;
   //找不到 x 和 m
}
```

《C++的变量、常量、程序空间》

- 正在运行中的程序所占用的内存空间：程序代码指令空间、变量空间、常量空间、程序动态申请的空间（new、malloc 等）。这些空间被组织(管理)为段:

➢ 代码段（用于存放程序的指令代码）、堆栈段（非静态局部变量、及其他用途）、数据段（用于全局变量和静

态变量)、常量段 (用于保存常量数据, 例如常量字符串)

- 不管程序以何种方式退出，代码段空间和常量段空间都会被释放。

指针。指针 (不管多少重) 也是变量，编译器会给指针变量分配内存单元。

- 指针变量是用来保存地址的，对于 win32 (x86) 系统，地址是 32 位 (4 个字节)，所以编译器会给指针变量分配 4 个字节的内存单元。不管多少重指针、也不管是什么类型的指针，由于指针本质上是一个地址，所以编译器会给不同的指针变量分配相同大小 (4 个字节)的内存单元。
- short x[10][20];
- short *y[10][20]; //怎么解释?
- short (*z)[10][20];
- x 是 1 个指向 10 个元素的数组，其中每个元素又包含 20 个元素，每个元素是 short 类型。所以 x 是 1 个 2 维 short 型的数组，x 变量占 10*20*2 个字节的内存。
- y 是 1 个指向 10 个元素的数组，其中每个元素又包含 20 个元素，每个元素是 1 个 short 类型的指针。所以 y 是 1 个 2 维 short *类型的数组，y 变量占 10*20*4 个字节的内存。
- z 是 1 个指针，指向 10 个元素的数组，其中每个元素又包含 20 个元素，每个元素是 short 类型。由于 z 是 1 个指针 (指向 1 个 2 维数组)，所以 z 变量占 4 个字节的内存。

(1) 局部空间 (堆栈段)

- 用于分配局部变量 (生命期是函数内部):
- 函数内部定义的非静态变量 (包括非静态 const 变量)、函数调用时的传入的实参、函数返回的对象
- 在堆栈段定义的变量都会被 C++释放:
- 函数内部的局部变量在函数返回时被释放；函数调用时的传入的实参和函数返回的对象，在主程序调用完函数后被释放。

```
class A { … };
int main() {
  const int k;
  A a;
  a = f(k, a); //k匹配吗?
/** 在堆栈中申请空间拷贝k
和a，调用完后将堆栈中临
时对象b拷贝到a，最后释放
堆栈中k,a,b的临时空间 **/
  return 0; //释放a和k空间
}
```

```
A f(int k, A a) {
  int x;
  const int j;
  A b = a;
  return b;
  /** 在堆栈中申请
  临时空间保存b,
  然后释放函数内部
  定义的x, j, b **/
}
```

(2) 全局空间 (数据段)

- 全局空间上定义的变量 (对象), 生命期是整个程序的生命期。
- 定义在全局空间的变量 (对象) :
- 非静态全局变量 (包括 const 对象), 不包括全局的 const 简单类型 (如 const int k)；静态变量 (局部和非局部), 不包括 static const 简单类型 (如 static const int k)；用指令申请的空间 (如 new、malloc 等) ；全局变量 (非静态)具有唯一性，作用域是整个程序。2 个.cpp 文件不能定义同名的全局变量。如果 1 个.cpp 文件需要访问定义在另外 1 个.cpp 文件的全局变量时，必须用 extern 声明。
- 非局部静态变量 (定义在函数外面的静态变量)，作用域是当前.cpp 文件。2 个不同.cpp 文件可以定义同名静态变量。非局部静态变量可以与其他.cpp 中的全局变量同名, 但访问不到同名的其他.cpp 中的全局变量。
- 局部静态变量 (定义在函数内部的静态变量), 作用域是当前函数内部(生命期是整个程序运行期间)。可以定义与非局部静态变量同名的局部静态变量。

(3) 常量空间 (const 段)

一些常量的值被存贮在该空间。定义在该空间的常量是不能修改的，若强行修改会引起程序崩溃。

定义在 const 段的 3 个典型常量 (变量):

- 常量字符串。类型是 const char * ，而不是 char *。字符串需要以 0 结尾，所以 “abc” 的长度是 3，需要 4 个字节的存贮空间；所有(全局、局部、类内)的静态 const 简单变量 (如 static const int k)；全局非静态 const 简单变量 (如 const int k)

- const char *p = “abc”;
- char c1 = p[1];
- char c2 = “abc”[1]; //???
- char c3 = “abc”[-1]; //???
- p[0] = ‘1’; //???
- (char *)p[0] = ‘1’; //???
- char *q = “abc”; //???
- char *q = (char *)“abc”; //???
- q[0] = ‘1’; //???
- (char *)“abc”[1] = ‘1’; //???
- char s[20]; strcpy(s, “abc”);
- s[3] = ‘1’; //???

```
const int x = 1;
struct  A {
        int  k;
        const int  i;
        static const int  j;
        A( ): i (-1) { }
} a;
const int  A::j = 2;
int main( ) {
      *(int *)&x = 0;          //语法正确，程序奔溃
      *(int *)&A::j = 0;    //语法正确，程序奔溃
      *(int *)&a.i = 0;        //正确, a.i = 0
}
```

《类内数据成员的缺省值》

- 在定义一个类时，一般可以设定类中数据成员的缺省值。但对于 static 数据成员，一般不能直接设定缺省值，一个例外的情况是“static const 整型变量”。 也就是说，可以在类内定义“static const 整型变量=整数”。
- 整型变量包括：bool、char、short、int、long、long long 及它们的 unsigned 型。
- 对于类内的“static const 整型变量”，一旦在类内设定了缺省值，这个静态变量也就创建了； 如果没有在类内设定缺省值，也没有在类外初始化，那么这个静态变量是不存在的。
- 对于类内的非 static 变量，即使设定了缺省值，也能重新赋值 (也可以在构造函数的初始化参数列表中重新初始化)。
- 类内数据成员，只能使用等于号(=)设定缺省值，不能直接使用圆括号的方式，即类内只能使用 int x=1，而不能使用 int x(1)。

```
class A { /**... **/};
class B {
   const A o = A();
   int  i = 0;
   int  &j = i;
   int  num[10] = { 1, 2, 3 };
   int  *const p = num;
   const int x = 1;
   volatile const float y = 1.0f;
   static int a;              //static非const的数据成员不能设定初始值
   static volatile int b;     //static非const的数据成员不能设定缺省值
   static const int c = 2;    //static const 的整型变量可以设定缺省值
   static const int d;        //可以设定d的缺省值
   static const float PI;     //static const 的非整型变量不能设定缺省值
   static const int e[2] = { 1, 2 }; //错, 静态const数组不能设定缺省值
   int y(1);                         //错, 只能 int y = 1;
};
int B::a = 1;
volatile int B::b = 1;
const float B::PI = 3.14159f;
int main() {
   cout << B::d;   //错, B::d 不存在
   cout << B::b << B::c;   //对
}
```

《函数的调用与返回》

```
class A {
      char *p;
public:
      A( ) {   p = new char [10]; }
      ~A( ) { if(p) delete p; p = 0; }
} a;
int f(A a)   { return 0; }
int main( ) {   f(a); }
```

——程序崩溃!

调用子程序时，编译器在堆栈中构建所需要的参数。

- 如果参数是指针（引用)类型，则将实参的地址拷贝到堆栈。
- 如果参数是简单类型的变量，则将实参的值拷贝到堆栈。
- 如果参数是对象（例如 A a），则调用 A(const A &) 在堆栈中构建对象，该临时对象在失去作用范围时将被析构。

```
struct A {
  A() { cout << "A() "; }
  A(const A &a) { cout << "A(a) "; }
  ~A() { cout << "~A() "; }
};
int f(A *a) { return 0; }
int g(A &a) { return 0; }
int h(A a)  { return 0; }
int main()
{
  A  a;
  A b = a;
  f(&a);
  g(a);
  h(a);
}
```

```
int main()
{
  A  a;       //A()
  A b = a;    //A(a)
  f(&a);      //
  g(a);       //
  h(a);       //A(a), ~A()
}             //~A(), ~A()
```

如果去除 A(const A &)，将怎样？

```
struct A {
  A() { cout << "A() "; }
  ~A() { cout << "~A() "; }
};
int main()
{
  A  a;       //A()
  A b = a;    //
  f(&a);      //
  g(a);       //
  h(a);       //~A()
}             //~A(), ~A()
```

子程序返回机制：

- 如果返回类型是 void，则不做任何事情。
- 如果返回类型是指针（引用)、简单类型(int、float 等)，则地址、简单类型的值保存到 EAX 寄存器中。主程序从 EAX 获取返回值。
- 如果返回 1 个对象（例如 A a），则调用 A(const A &) 在堆栈中构建 1 个临时对象。主程序调用 operator=(const A &)将该临时对象赋值给变量，然后析构该临时对象。

```
struct A {
  A() { cout << "A() "; }
  A(const A &a) { cout << "A(a) "; }
  ~A() { cout << "~A() "; }
  A &operator=(const A &a) {
     cout << "=() "; return *this; }
};
int f(A *a) { return 0; }
int g(A &a) { return 0; }
A h(A a)  { return a; }
```

```
int main()
{
  A  a;
  A  b = a;
  f(&a);
  g(a);
  b = h(a);
  A  c = h(a);
}
```

```
int main()
{
  A  a;          //A()
  A  b = a;      //A(a)
  f(&a);         //
  g(a);          //
  b = h(a);      //A(a), A(a), ~A(), =(), ~A()
  A c = h(a);    //A(a), A(a), ~A()
}                //~A(), ~A(), ~A()
```

《虚函数与多态》

(1) 虚函数定义

用 virtual 定义的成员函数（虚函数必须是类的实例成员函数，即有 this 指针的函数)。

```
class A {
int k;
public:
int f( ) { return k; }
virtual int g( ) { return 1; }
virtual static int h( ); //error，不是实例成员函数
A(int k) { this->k = k; }
};
```

(2) 虚函数作用

在类的继承链中，实现动态多态。

- 用基类对象指针（引用)指向类型派生类对象，通过这个基类指针(引用)去调用虚函数，就可实现动态多态(基类和派生类中定义了函数原型相同的虚函数，运行时确定调用哪一个函数)。
- 虚函数只有在继承关系时才起作用。

(3) 虚函数的继承性

- 一旦基类定义了虚函数，即使没有 virtual 声明，所有派生类中原型相同的非静态成员函数自动成为虚函数。
- 构造函数构造对象的类型是确定的，不需根据类型表现出多态性，故不能定义为虚函数。
- 析构函数可通过基类指针(引用)调用，基类指针指向的对象类型可能是不确定的，因此析构函数可定义为虚函数。

(4) 虚函数的多态性

假定如下的继承关系：C0 ← C1 ← … Ck ← … ← Cn　　即，C0 是祖先类，Cn 是子孙类。有如下语句：

- Ck c; C0 *p = (C0 *)&c; p->f( ); int k = p->i;
- 或者：C0 &q = c; q.f( ); int k = q.i;

对于语句 p->f()，编译器将会做如下工作：

(a)在 C0 类寻找函数 f()，如果没有找到或找到但 f()不能访问，则报错；

(b)如果 C0 中有 f()且可以访问，这时判断 f()是否是虚函数，若不是虚函数则直接调用 C0::f()；若 f()是虚函数，则转下一步；

(c)沿着 Ck 到 C0 的方向，查找虚函数 f()，只要发现某个类 Cm (0<=m<=k)中重定义了 f() (即使 Cm::f()是 private 属性)，则调用 Cm::f()。

数据成员没有虚特性（没有多态性)

- 对于语句 p->i，编译器在 C0 类寻找变量 i，如果找到了且可以访问则直接使用；如果没有找到或找到了但 i 不能访问，则报错。
-

《类的存贮空间》

1. 简单类的存贮空间
2. 无虚函数的派生类存贮空间

(1) 只有普通数据成员才占空间

(2) 静态数据成员和函数成员不占空间

```
class A {
        static long a;
        int  i;
public:
        int  k;
        static int b;
        friend int h() { return  -1; }
        int f() { return 0; }
        static int g() { return 1; }
        A() { }
} x;
```

x的存贮空间

int i
int k

Question: 能否将h()写成 { return k; } ?

派生类对象需要把基类空间包含进去

```
class A {
        static long a;
        int i;
public:
        int k;
        static int b;
        int f() { return 0; }
        static int g() { return 1; }
};
class B : A {
    int  i, j, k;
    static  int m;
} x;
```

x的存贮空间

int i int k	A	B
int i int j int k		

多继承和虚基类怎么处理？

如何访问x中A的数据成员？

3. 访问对象 (无虚函数)中的数据成员

4. 有虚函数的简单类存贮空间

```
class A {
   static long a;
   int  i;
public:
   float  k;
   static int b;
   int f() { return 0; }
   static int g()
   { return 1; }
   A(int x,int y)
   { i = x; k = y+0.5f; }
};
```

```
class B : A {
   int  i, j;
public:
   int  k;
   static int m;
   B(int x, int y, int z): A(x-1,y-1)
   { i = x; j = y; k = z; }
};
void main()
{
   B  b(1,2,3);
   int    *p1 = (int *)&b;
   float *p2 = (float *)(p1+1);
   int    *p3 = (int *)(p2+1);
   cout << p1[0] << p2[0] <<
   p3[0] << p3[1] << p3[2];
}
```

Results: 0 1.5 1 2 3

偏移 00 - 03：虚函数表VFT的地址
偏移04开始：非static的数据成员

```
class A {
   int  i, j;
   static  int  x, y;
   virtual void f( ) {  cout << "f()"; }
public:
   int  k;
   virtual void g() { cout << "g()"; }
   int h() { return i + k; }
};
```

//假设 sizeof(int) = 2

类A的存贮空间

偏移	内容
00-03h	VFT地址
03-04h	int i
05-06h	int j
07-08h	int k

VFT

偏移	内容
00-03h	f()的入口地址
04-07h	g()的入口地址

5. 有虚函数的派生类存贮空间

(1) 基类没有虚函数(派生类有)

```
class A {
   int  i;
   static  int  x, y;
public:
   int  k;
   int  h() { return i + k; }
};
class B: public A {
   int  i;
public:
   int  k;
   int  h() { return i + k; }
   virtual void f()  { cout << "B::f()"; }
   virtual void g() { cout << "B::g()"; }
};
```

//假设 sizeof(int) = 2

类B的存贮空间

偏移	内容
00-01h	int A::i
02-03h	int A::k
04-07h	B的VFT地址
08-09h	int B::i
0A-0Bh	int B::k

B的VFT

偏移	内容
00-03h	f()的入口地址
04-07h	g()的入口地址

(2) 基类有虚函数

```
class A {
   int  i;
   static  int  x, y;
public:
   int  k;
   virtual void f()  { cout << "A::f()"; }
   virtual void g() { cout << "A::g()"; }
};
class B: public A {
   int  i;
public:
   int  k;
   int  h()  { return i + k; }
   virtual void f()  { cout << "B::f()"; }
   virtual void k() { cout << "B::k()"; }
};
```

//假设 sizeof(int) = 2

类B的存贮空间

偏移	内容
00-04h	B的VFT地址
00-01h	int A::i
02-03h	int A::k
08-09h	int B::i
0A-0Bh	int B::k

B的VFT

偏移	内容
00-03h	B::f()的入口地址
04-07h	A::g()的入口地址
08-0Bh	B::k()的入口地址

派生类 VFT 构造方法:

(1) 将基类 A 的 VFT 复制到派生类 B 的 VFT 的开始处;

(2) 若派生类 B 对基类 A 的某个虚函数 f()进行了重定义，则在刚拷贝的基类 VFT 中，用 B::f()的地址替换 A::f()函数的入口地址;

(3) 若在派生类 B 中定义了新的虚函数，则依次将这些新的虚函数的入口地址尾加到 B 的 VFT。

6. 利用存贮空间访问变量和虚函数

```
class A {
   int  i;
public:
   virtual void f()
   { cout << "A::f \n"; }
   virtual void g()
   { cout << "A::g \n"; }
   A(int x) { i = x; }
};
class B : public A {
   int  i, j;
public:
   virtual void h()
   { cout << "B::h \n"; }
   void g()
   { cout << "B::g \n"; }
   B(int x,int y): A(x-1)
   { i = x; j = y; }
};
```

```
void main()
{
   B  b(1,2);
   int *v = (int *) ( (char *)&b +
            sizeof(void *) );
   printf("A::i=%d, B::i=%d,
   B::j=%d \n", v[0],v[1],v[2]);
   void **vfb = *(void ***)&b;
   for(int i = 0; i < 3; i++)
   {
      void (*f)() = (void (*)())vfb[i];
      f();
   }
}
```

A::i=0, B::i=1, B::j=2
A::f B::g B::h

能否直接取虚函数的地址, 如 &b.f ？

《名字空间》

- 可以在不同的模块中定义同名的名字空间。编译器会将不同模块中定义的同名的名字空间整合为一个空间。
- 每个模块中名字空间内的符号，作用域是当前模块。
- 引用其他模块中同名的名字空间内的符号时，需要在本模块的名字空间中用 extern 声明。

- 用 using 引用名字空间的符号时，只能引用本模块的名字空间中的符号。

```
//1.CPP
#include <iostream>
using namespace std;
namespace A {
   int  x = -1;
   int  f( ) { return 1; }
}
using namespace A;
//访问不到2.cpp中A的符号
int main( )
{
   cout << h();          //error
   cout << y + A::y;  //都访问不到
}
```

```
//2.CPP
#include <iostream>
namespace A {
   int  x;  //error
   int  y = 2;
   int  g() { return -2; }
   //extern int f();
}
using namespace A;
//访问不到1.cpp中A的符号
using A::f;
//error, 只能引用本模块A中的符号
int h() { return f(); }
```

- 名字空间（包括匿名名字空间）可以分多次定义。

```
//1.CPP
namespace A {
      int x = -1;
      int f() { return 1; }
}
using namespace A;
namespace A {
   int y = -1;  //error, 重复定义
   int g() { return 2; }  //error, 重复定义
   int h() { return 0; }
}
int main() {
      f();   h();
      g();  //error, 访问不到2.CPP的g()
      cout << y;  //error, 访问不到
}
```

```
//2.CPP
namespace A {
      int y = 2;
      extern int f();
}
using namespace A;
namespace A {
      int g() { return -2; }
}
int f( ) { return 1; }
void m() {
      h();   //error,访问不到
      f();   //error,全局f()? A::f()?
      ::f();  //全局f()优先
      A::f();
}
```

(1) using namespace 名字空间名称;

本模块可以访问这个名字空间中的所有符号（即使该名字空间是多次定义的）。可以在当前模块中再定义和名字空间中同名的标识符。

(2) using 名字空间名称::成员名称;

- 只能在本模块中访问这个成员。编译器将该成员的定义加入当前模块，不能在当前模块中再定义和该成员同名的标识符。
- 引用的成员必须在用 using 引用前已经声明（即使还没有定义也可以）。

```
//1.CPP
namespace A {
      int y = 2;
      int g() { return -2; }
      extern int f();
}
using namespace A;
int x = 11;    int y = 22;
int f() { return 1; }
int g() { return 2; }
void m() {
   h();    //error, 访问不到
   f();    //error, 全局f()? A::f()?
   g();    //error, 全局g()? A::g()?
   ::f();  //全局f()优先
   ::g();  //全局g()优先
   A::f();   ::y++;   A::y++;
}
```

```
//2.CPP
namespace A {
      int x = -1;
      //int y = -2; //error,重复定义
      int f() { return 1; }
      int h() { return 0; }
      //int h(int);
}
using A::h;
using A::x;
float x = 0;  //error, 重定义
namespace A {
   int h(int x) { return x; }
}
int main() {
   h(1);  //error, 没有定义，重新在此之前
      using namespace A或using A::h即可
   //将 using A::h 改为 using namespace A?
}
```

《深入理解成员指针》

(1) 定义实例成员指针

实例数据成员指针：变量类型　类名::*变量名；

实例函数成员指针：返回类型　(类名::*变量名) (参数原型)；

- 实例数据成员指针的值是数据成员（相对对象首地址）的偏移值
- 实例函数成员指针实际上是函数的物理地址

```
struct A {
    int j, i;
    int f(int x) { return i + x; }
    int A::*u;
    int (A::*pf)(int);
};

int main( ) {
   A a;
   a.u = &A::i;                  //a.u = 4
   a.pf = &A::f;                 //a.pf = A::f( )的物理地址
   int A::*p = &A::i;            //p = 4
   int (A::*ff)(int) = &A::f;    //ff = A::f( )的物理地址
};
```

(2) 使用实例成员指针

必须结合对象(引用)使用 .*、或结合对象指针 使用 ->*，来访问对象中的成员，用法是：

- 对象.*数据成员指针；对象指针->*数据成员指针；(对象.*函数成员指针) (参数)；(对象指针->*函数成员指针) (参数)

```
struct A {
   int j, i;
   int f(int x) { return i; }
   int A::*u;
   int (A::*pf)(int);
} a;
```

```
int main( ) {
   A *p = &a;
   a.u = &A::i;     //a.u = 4
   a.pf = &A::f;    //a.pf=A:f()的物理地址
   int A::*u = &A::i;  //u = 4
   int (A::*ff)(int) = &A::f;  //ff=物理地址
   int i = a.*u;  //注意区分，此处为a外的普通u
   i = p->*u;
   i = a.*a.u; //此处为a内的a.u
   i = p->*a.u;
   int x = (a.*ff)(1);
   x = (p->*ff)(2);//a外
   x = (a.*a.pf)(3); //a内
   x = (p->*a.pf)(4);
};
```

(3) 实例成员指针说明：不能移动，不能参与运算，不能强制类型转换。

```
struct A {
   int j, i;
   int f(int x) { return i; }
   int A::*u;
   int (A::*pf)(int);
} a;
```

```
int main( ) {
   int A::*pi = &A::i;
   int (A::*pf)(int) = &A::f;
   pi++;  //错, 不能移动
   pf++;  //错, 不能移动
   pi = (int A::*)i;    //错, 不能强制转换
   int k1 = pi + 1;     //错, 不能参与运算
   int k2 = (int)pi;    //错, 不能强制转换
   int k3 = (int)pf;    //错, 不能强制转换
   int (*f)() = &main;
   k3 = (int)f;  //对, 普通指针可以强制转换
   float i = (float)(a.*pi);      //对
   float j = (float)(a.*pf)(1);  //对
};
```

- 静态成员指针本质上是普通指针，需要以普通指针的形式去定义和访问，不能使用“类名::*变量名”实例成员的方式去定义和访问。
- 静态成员指针的值是静态成员的物理地址。

```
struct A {
    int i, j;
    static int k;
    int f(int x) { return 0; }
    static int g(int x) { return x; }
} a;
int A::k = 1;
```

p++对，因为一个int为4字节，大小确定
f++错，static函数代码段大小不确定，
加了之后不知道会到哪里

```
int main( ) {
    int A::*pi = &A::k;          //错
    int (A::*pf)(int) = &A::g;   //错
    int *p = &A::k;
    int (*f)(int) = A::g;
    float *q = (float *)f;       //对
    int k1 = (*f)(100);
    p++;                         //对
    q++; q + 1;                  //对
    f++;                         //错
    int k2 = f + 1;              //错
    int k3 = (int)f + p[2];      //对
};
```

- 类的函数成员的代码是不依赖于任何对象而独立存在的，实例函数成员指针实际上是成员函数的物理地址。
- 能否不通过对象而直接调用实例函数成员呢？

(不通过“(a.*f)(实参) ”的形式去调用实例成员函数)

```
struct A {
  int a = 1;
  int f() { return 1; }
  int g(int x) { return a + x; }
} a;

//不通过对象去调实例函数成员
int main( )
{  int (A::*h)() = &A::f;
   int k = (a.*h)();
   int (*p)(void *);
   p = (int (*)(void *))h;  //error
   k = (*p)((void *)0);  //collapse
}
```

实例函数成员指针实际上是物理地址，但不能进行强制类型转换和指针运算！

```
//不通过对象去调实例函数成员
int main( )
{  int (A::*h)() = &A::f;
   int k = (a.*h)();
   int (*p)(void *);
   //p = (int (*)(void *))h; //error
   _asm mov eax, h
   _asm mov p, eax
   k = (*p)((void *)0);  //k=1
}
```

```
struct A {
  int a = 1;
  int f(int x)
      { return 2 * x; }
  int g(int x)
      { return a + x; }
} a;
```

不通过对象去调
实例函数成员

```
int main() {
    int (A::*pf)(int) = &A::f;
    int (A::*pg)(int) = &A::g;
    int (*f)(void *, int);
    int (*g)(void *, int);
    //f = (int (*)(void *, int))pf;   //error
    //g = (int (*)(void *, int))pg;  //error
    _asm mov eax, pf
    _asm mov f, eax
    _asm mov eax, pg
    _asm mov g, eax
    int k1 = (*f)((void *)0, 10);   //k1 = 20
    int k2 = (*g)((void *)0, 10);   //collapse
    int k3 = (*g)(&a, 10);  //k3 = 11
}
```

《static_extern_friend 等限定符》

static, extern　指明存贮模型，修饰变量和函数, static 和 extern 不能同时使用
const, volatile　指明属性，修饰变量和函数
mutable 只能用在类里面修饰非 const 的实例数据成员
virtual　只能用在类里面修饰实例函数成员
friend　修饰一个函数(普通函数、成员函数)
inline　修饰一个函数 (普通函数、成员函数)

- 对于用 static 声明的普通函数，在定义函数体时 static 可以省略 (也可以不省略);
- 对于用 static 声明的类内成员函数，在类外定义函数体时一定省略 static (像初始化类内定义的 static 变量一样)。
- static 和 extern 不能连用
- 只有对于 [static] const 变量，先声明后定义，才有意义 (先定义后声明 ⇔ 只定义不声明)，表明将该变量的作用域修改为整个程序。

```
例：extern const   int z;        //其他模块可以访问 z
static const int z = 3;
static void h();
struct A {
        static const int k;       // = 1 ?;
        static void f();
        friend static void h();
        //extern friend static void h();     //error
};
const int A::k = 10;
void A::f() { }
//static void A::f() { }   //error
void h() { }   //OK
//static void h() { }   //OK
int main() { return 0; }
```

- static/extern 互斥(不能连用)，可以与 const, volatile, inline, friend 相互连用； 不能与 virtual, mutable 连用.
- const, virtual, friend, inline 可以相互连用.
- mutable 只能与 volatile 连用，其他限定词都不能与 mutable 连用.